AD-A040 989 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
A SEMANOL (76) SPECIFICATION OF MINIMAL BASIC.(U)
MAY 77 P T BERNING F30602-76-C-0245
UNCLASSIFIED RADC-TR-77-170-VOL-1 NL

1 OF 1
AD
A040989

END
DATE
FILMED
7-77

12

NW

RADC-TR-77-170, Volume I (of two)
Final Technical Report
May 1977

A SEMANOL (76) SPECIFICATION OF MINIMAL BASIC

TRW Defense and Space Systems Group

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

APPROVED: John M. Ives

JOHN M. IVES, Captain, USAF
Project Engineer

APPROVED: [illegible]

ALAN R. BARNUM
Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

19 **REPORT DOCUMENTATION PAGE**	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 RADC-TR-77-170, Volume I (of two) / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* 6 A SEMANOL (76) SPECIFICATION OF MINIMAL BASIC.	5. TYPE OF REPORT & PERIOD COVERED 9 Final Technical Report, 27 Apr 76 — 27 Jan 77,
	6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) 10 Paul T. Berning	8. CONTRACT OR GRANT NUMBER(s) 15 F30602-76-C-0245 new
9. PERFORMING ORGANIZATION NAME AND ADDRESS TRW Defense and Space Systems Group One Space Park Redondo Beach CA 90278	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 16 55500846 17 08
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441	12. REPORT DATE 11 May 1977
	13. NUMBER OF PAGES 50 12 53p.
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Same	15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
Captain John M. Ives (ISIS)

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

SEMANOL, SEMANOL (73), SEMANOL (76), BASIC, Minimal BASIC, language definition, standardization, semantics, syntax, language control, metalanguage, interpreter

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report describes the work performed and the results achieved in the preparation of a SEMANOL (76) metalanguage specification of the Minimal BASIC programming language. It explains the formal specification of Minimal BASIC produced, the SEMANOL (76) system used for the specification, and the ambiguities found in the Minimal BASIC draft proposal as a consequence of preparing this formal specification. The report also includes a suggested approach to formalizing machine dependent aspects of arithmetic and source language optimization; unfortunately, a lack of time did not permit these methods to be

over

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

409637

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

applied to Minimal BASIC. The SEMANOL specification of Minimal BASIC was extensively computer tested with the SEMANOL (76) Interpreter; an operational specification of Minimal BASIC was thus created.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

EVALUATION

Since so many non-compatible dialects have been written calling themselves the BASIC programming language, the ANSI X3J2 committee is defining a common core, that is, a Nucleus, that must be included in valid BASIC interpreters and compilers. However, the X3J2's fundamental draft specification consisted of English explanations of the programming language's semantics as addendums to the more formalized Backus-Naur format of the syntax.

The USAF supports the stabilization and formulation philosophy of this committee. To provide the desired integrity of the nucleus specification and, at the same time, to exercise SEMANOL, a specification language under development, the Air Force contributed an evaluation and formalized specification of the recommended X3J2 draft specification semantics By defining the BASIC Nucleus in SEMANOL, numerous potential inconsistencies and discrepancies in the English-type specification were quickly and efficiently discovered.

[illegible]

JOHN M. IVES, Captain, USAF
Project Engineer

INTRODUCTION

The project reported upon herein had as its objective the preparation of a formal SEMANOL specification of the Minimal BASIC programming language as given by the X3J2/76-01 draft proposal of the American National Standards Committee. The X3J2 draft proposal was of a conventional form and so relied heavily upon the use of prose text. Consequently, it was recognized that it was likely to be incomplete and that many parts would very likely be subject to varying reader interpretations. The preparation of a SEMANOL specification could then be expected to reveal such ambiguities, as well as to provide a formal specification document that would be complete, precise, and uniformly understood. It was also intended that the SEMANOL specification of Minimal BASIC be operable with a SEMANOL Interpreter program upon the HIS-6180 Multics computer system; the formal specification could then be computer tested and otherwise be available for use in programming language development activities.

These objectives were all successfully met. A formal SEMANOL specification of Minimal BASIC was written in the SEMANOL(76) metalanguage. SEMANOL(76), developed under concurrent contract F30602-76-C-0238, is the newest version of the SEMANOL metalanguage, and its use meant that the readability of the resulting Minimal BASIC specification was thereby improved. The use of SEMANOL(76) also allowed the newest version of the SEMANOL Interpreter program to be used, with an accompanying substantial improvement in processing efficiency and user convenience. The specification was completed and was thoroughly tested with the SEMANOL(76) Interpreter in the performance period; an operational specification of Minimal BASIC now exists.

The preparation of the SEMANOL(76) specification of Minimal BASIC led to the discovery of many instances of seemingly incomplete, unclear, or contradictory description in the X3J2 draft proposal. These cases were reported to Rome Air Development Center (RADC) in three reports, and hence to the X3J2 committee; many of these items are repeated later within this report. While the writing of a SEMANOL(76) specification could be expected to reveal some problems, it is felt that the sensitivity of the TRW group doing this analysis to the subtleties of language design and description served to yield a more comprehensive list of ambiguities than would normally be expected.

The SEMANOL(76) specification of Minimal BASIC produced in this project can be useful in several ways. Since the SEMANOL(76) notation is a special sort of programming language that can be processed by the SEMANOL(76) Interpreter, the specification can be computer tested until agreement is reached that the specification accurately reflects its designers' intentions. Great confidence can thus be gained that Minimal BASIC is properly defined. This form of specification then provides:

1. A specification that can be uniformaly understood by those who read it. This specification precision will make clear what Minimal BASIC is, and so will allow discussion about the language to be clearly and precisely expressed. We believe that the undeniable precision of SEMANOL(76) retains a degree of readability that is adequate even for those having only a general familiarity with the SEMANOL(76) metalanguage; this readability results from the intuitive clarity provided by the use of the SEMANOL(76) metalanguage and well developed conventions for its application.
2. A description of Minimal BASIC which can be used in the development of language processors built to unambiguous and complete specifications.
3. A means of directly testing language processor conformance to that specification. Testing itself must still be done through the use of a carefully constructed set of test programs written in Minimal BASIC. However, the correct result of executing these tests is determined by use of the SEMANOL(76) Interpreter program, and it is these results against which language processors will be measured. This test procedure is especially attractive since implementation dependent semantics, including machine dependencies, can be part of the SEMANOL(76) specification.
4. A good means for testing the effect of proposed language changes. Such changes can be described in SEMANOL(76) and the currently accepted language specification modified to include the proposal. The descriptive process itself should reveal the scope and implications of the change, while testing through use of the Interpreter will allow the effect of these changes to be verified. Changes can thus be fully understood before they are adopted.

5. Assistance in preparing programming manuals which are consistent with language processor implementations. Although it is not expected that the SEMANOL(76) specification of Minimal BASIC will be used by most programmers as their language reference document, a conventional programming manual can be prepared from the same SEMANOL(76) descriptive foundation as the language processor with the expectation that the two will closely match. In addition, the SEMANOL(76) specification can be used in programming organizations as a reference document to answer the subtle and difficult questions about Minimal BASIC that the conventional manual fails to treat adequately.

The remainder of this report describes the way in which this project was performed, presents many of the ambiguities found in the X3J2 draft proposal, describes the SEMANOL(76) specification of Minimal BASIC that was delivered, and includes a brief introduction to the SEMANOL(76) system of semantic description. A discussion of a promising approach to formalizing optimization and machine effects is also given; this investigation went somewhat beyond the expected scope of performance, but was prompted by the question of "conformance" as it was given in the X3J2 draft proposal.

PROJECT PERFORMANCE

The work described in this report was performed in the nine month period of May 1976 through January 1977. While much of the work was similar to that which would be involved in writing a conventional computer program, the special nature of writing a formal specification of a programming language meant that some of the activities were somewhat unconventional.

The central task to be performed in this project was the preparation of a formal SEMANOL metalanguage specification of the Minimal BASIC programming language. The source document used in performing this task was the "Proposed American National Standard for Minimal BASIC" of January 1976 (X3J2/76-01). The X3J2 document described Minimal BASIC in terms of a conventional context-free syntax notation and an English prose description of syntactic constraints, semantic actions, and certain error treatments. The document augmented the language presentation with a summary of committee discussions and votes. Generally speaking, this document seemed to be carefully done; nevertheless, it presented many problems to arriving at a clear understanding of what Minimal BASIC was meant to be.

A preliminary analysis of the Minimal BASIC draft proposal was done first to formulate the general design approach to be followed and to identify language features that were unclear. This analysis led to the construction of lists of (1) implementation dependent parameters, (2) error conditions and responses, and (3) non-context-free restrictions to be enforced. These lists were derived from the draft proposal and frequently, particularly for implementation parameters, represented extensions to the explicit items of the draft proposal. These extensions reflected the needs of our planned implementation, as well as sometimes providing detail that seemed to be missing otherwise. Additionally, a list of questions about the draft proposal was prepared. This list of suspected ambiguities was compiled with great care, and items were included only after an intensive search of the Minimal BASIC document failed to provide a resolution.

The organization and content of the formal specification that was delivered are described later in this report. The writing of that specification began with the creation of a context-free grammar; this grammar corresponds very closely with that of the draft proposal (although notationally somewhat different, of course). The semantics of high level interpretation control were done next, followed by description of the semantics of the various statement types, standard naming conventions, and input-output. Floating-point evaluation was done independently and later integrated into the specification. As parts of the specification were written, they were processed by the Translator until they were syntactically correct. As complete units were developed, they were processed by the Executer (equivalent to executing traditional programs) with simple test cases. Code writing and testing overlapped,and emphasis was given to the early creation of an operational specification that would be capable of driving the interpretation of very simple Minimal BASIC programs. The fuller specification was then developed incrementally.

The computer used in this project was the HIS-6180 Multics system located at Rome Air Development Center. The computer programs (i.e., the Translator and Executer) used to process the formal specification were actually part of two different SEMANOL systems. The SEMANOL(73) system, developed earlier, was operational upon Multics when this project began. However, we were engaged in a concurrent project to improve the SEMANOL(73) system in contract F30602-76-C-0238. The concurrent project was planned to produce an improved metalanguage, SEMANOL(76), and more efficient versions of the Translator and Executer. The specification of Minimal BASIC was thus written initially in SEMANOL(73) in the expectation that it would later be converted to SEMANOL(76). Most testing was done by use of the SEMANOL(73) Translator and Executer, and the test results were then confirmed after the specification of Minimal BASIC was converted to the SEMANOL(76) metalanguage. Fortunately, the SEMANOL(76) Translator and Executer were working well when needed by this project. The conversion to SEMANOL(76) also went very well because it was anticipated and because the two metalanguage dialects are, after all, not radically different. Note that the context-sensitive constraints for Minimal BASIC were written directly in SEMANOL(76), due to SEMANOL(76)'s greatly improved facilities for stating such conditions, and so constituted an exception to this general practice. While this conversion took some small effort, it produced a specification of Minimal BASIC that has the visual advantage of improved SEMANOL(76) metalanguage readability and the operational advantage granted by the faster SEMANOL(76) Interpreter. It is therefore, a better specification than it would have been if it had been done with the SEMANOL(73) system.

While the early analysis of the X3J2 proposed standard revealed some ambiguities, the detailed specification design process and actual coding exposed others (and sometimes even clarified items found before that had seemed to be problems). A list of ambiguities found early in the performance period was delivered to RADC on 30 August 1976; it was then forwarded to the X3J2 committee by RADC as a public comment on the draft proposal. This original list was extensively enlarged through further analysis and the enlarged list sent to RADC on 19 November 1976. This second list of ambiguities also contained our intended solution for each case, and so allowed RADC to evaluate the acceptability of our interpretations. Many of these ambiguities and our reaction to each are presented later in this report. The revised X3J2 Minimal BASIC specification of December 1976 was also reviewed to determine which ambiguities of the earlier draft proposal had been resolved and which had not. A careful reading of the December 1976 document, but certainly not an intensive study, was also conducted to determine if new ambiguities had been introduced (a few were found). The results of this work were given to RADC in January 1977. Note that the December 1976 revision was otherwise ignored by this project because of its late arrival. In summary, a diligent effort was conducted to find definitional problems in the X3J2 draft proposal and to report these to RADC in a timely fashion. Furthermore, proposed solutions for each problem were made to RADC so that RADC participation in each TRW solution was insured. The ability of a SEMANOL application to find problems in conventional specification methods was again evident in this performance.

One special quality colored the way in which the SEMANOL(76) coding of the Minimal BASIC specification was done, and that is the desire for readability. A SEMANOL(76) program is to be read by people, and readability is shaped at least as much by style as it is by the language used. Past projects had established an organization and set of conventions that were felt helpful; these were applied here, but they offered only a general outline since each programming language is somewhat unique. For Minimal BASIC, the major problem was how best to differentiate between implementation independent features of the language and those that are meant to be determined by a language processor. The X3J2 draft proposal considered implementation parameters, if incompletely, and so provided rare assistance in this regard. However, many parameters were overlooked there and needed to be recognized during the specification. There are also details of implementation, such as those dealing with arithmetic, that must be supplied in a SEMANOL(76) specification, but which were a form of overspecification to the X3J2 committee. We have tried to collect all such parameter definitions and implementation specific semantics into a single section of the specification, and then to write the remainder of the specification so that these implementation dependent definitions could be easily recognized. Furthermore, it should be relatively easy for the reader to ignore these details while still gaining the higher level meaning of the semantics. This interest, in having a specification structure that can be read to the depth wanted by the reader, has affected all parts of the specification.

One virtue of the SEMANOL(76) system is that a specification of a programming language that is written in the SEMANOL(76) metalanguage can be computer tested by use of the Interpreter. The testing of the Minimal BASIC specification delivered in this project was considered relatively thorough, with arithmetic expression evaluation semantics being tested especially well. The testing was done with small Minimal BASIC programs, each of which normally dealt with a particular form of statement in a given context. Testing of complicated programs did not appear useful in this project and so was not done. All parts of the Minimal BASIC programming language were tested, including samples of all error conditions and context-sensitive syntactic restrictions. Well over one hundred test programs were successfully processed in this test phase. While it would be foolish to claim that the resulting specification is without error, we do feel it has been adequately tested and so is a sound product.

It should also be noted that performance upon this project raised various questions about formal semantics and the way they might be treated within SEMANOL(76) theory and practice. In the case of Minimal BASIC, particular attention was given to exploring methods by which implementation dependencies might be precisely characterized without being considered overspecified. It was felt that practical standards would certainly benefit by being able to include a characterization more useful than "implementation defined". Minimal BASIC is simpler than most programming languages and had been described in the draft proposal with some regard for these matters; it thus seemed to be a likely candidate for study in this regard. The method developed through our study appears in the section on Formalization of Implementation Dependencies in this report. The general method and its specific application to Minimal BASIC were worked out in moderate detail, even to the extent that the specification delivered is written to accommodate a standardized description of computer arithmetic. Unfortunately, this investigation had not been foreseen and so it could not be pursued further in this contract.

In summary, this project succeeded. It produced a SEMANOL(76) specification of Minimal BASIC. The specification is readable and well tested. Ambiguities in the X3J2 draft proposal were discovered, reported to RADC, and resolved for purposes of our specification. Beyond this, a promising method for formally dealing with implementation dependencies was devised and reported upon.

AN INTRODUCTION TO SEMANOL(76)

SEMANOL(76) is intended for use in describing (procedural) programming languages. A specification written in the SEMANOL(76) metalanguage is meant to provide an exact and complete definition of a programming language that is comprehensible to a suitably trained reader. That is, SEMANOL(76) is designed to supply people with a basis for communication about programming languages that is more precise than commonly employed description methods. Additionally, the formality of the SEMANOL(76) metalanguage permits operational use of the specification to be made upon a computer.

The specification method adopted is algorithmic. This choice stems from a feeling that the semantics of programming languages ought to be explained in this way. That is, semantics are concerned with explaining how something happens and not just in characterizing an input-output relationship. Certainly this is the way in which designers, compiler writers, and application programmers generally view the semantics of a programming language. Having a direct correspondence between the formal, operational, SEMANOL(76) expression of language semantics and a reader's intuitive conception of a language yields a specification method that can be easily understood. An algorithmic method also permits language details, such as those specific to a given implementation, to be described exactly when desired.

The SEMANOL(76) method considers a programming system, S, to be defined by $S = (P, I, T, \Phi)$ where

P = The set of programs which can be expressed in the programming system.

I = The set of input values.

T = The set of output traces. The trace is an ordered record of significant actions (such as assignment) that are performed by the program as it is executed; it is the visible manifestation of performing the algorithm that is the operational SEMANOL(76) specification of semantics. The trace is thus similar to a state sequence.

Φ = The semantic operator. This operator, given as $\Phi : P \times I \rightarrow T$, is considered to define the "meaning" of a program.

P, I, and T are each sets of strings which are specified by Φ, and whose individual members will be denoted by the corresponding lower case letters (i.e., $p \varepsilon P$, $i \varepsilon I$, $t \varepsilon T$.). The effect of sxecuting a given program, p, can then be denoted in terms of the semantic operator by

$$\Phi(p,i) = t$$

Thus Φ specifies the trace produced by any program in the system when that program is executed with any input value sequence. The SEMANOL(76) metalanguage is used for programming the semantic operator, thereby providing a method for formal specification of a programming language. Since the SEMANOL(76) metalanguage is itself a programming language, it also belongs to a programming system. To differentiate between these two systems, we will use the subscript j to identify elements of the programming system being defined by a SEMANOL(76) program (e.g., Minimal BASIC) and the subscript s to identify elements of the SEMANOL(76) system. The semantics of a Minimal BASIC program, p_j, are then expressed by

$$\Phi_j(p_j,i_j) = t_j$$

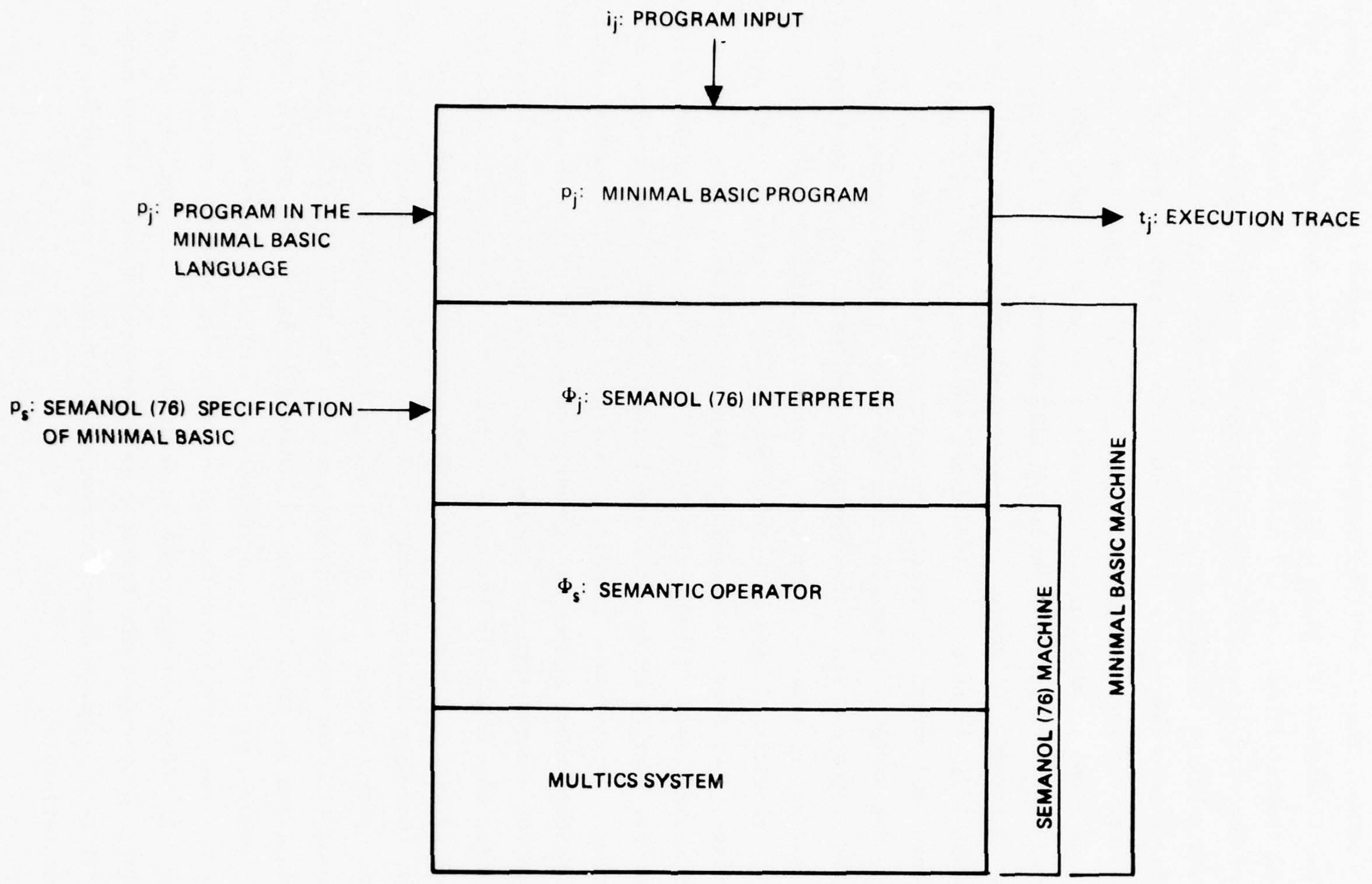
The semantic operator for Minimal BASIC, Φ_j, is expressed as a SEMANOL(76) program, p_s, which in turn is interpreted by a semantic operator for the SEMANOL(76) programming system, Φ_s.

Thus we have

$$\Phi_s(p_s,(p_j,i_j)) = \Phi_j(p_j,i_j) = t_j$$

and a formal definition of Minimal BASIC is provided by p_s. The SEMANOL(76) semantic operator, Φ_s, is defined in the SEMANOL(76) Reference Manual and has been implemented by the SEMANOL(76) Interpreter computer program.

This general view of language definition is shown graphically in Figure 1. As shown there, these levels of semantic specification correspond to defining a virtual machine for SEMANOL(76) and, based on that, one for Minimal BASIC.



i_j: PROGRAM INPUT
p_j: PROGRAM IN THE MINIMAL BASIC LANGUAGE
p_s: SEMANOL (76) SPECIFICATION OF MINIMAL BASIC
p_j: MINIMAL BASIC PROGRAM
Φ_j: SEMANOL (76) INTERPRETER
Φ_s: SEMANTIC OPERATOR
MULTICS SYSTEM
t_j: EXECUTION TRACE
SEMANOL (76) MACHINE
MINIMAL BASIC MACHINE

Figure 1: The SEMANOL(76) System

As observed, the SEMANOL(76) metalanguage is meant to describe semantic operators. Consequently, it is a high level language designed for the specific purpose of completely and exactly describing the syntax and semantics of procedural programming languages. The metalanguage permits high-level expressiveness and makes no special effort to minimize the primitives available. It thus contains some redundancy in its primitives and permits syntactic variations where this can aid reader interpretation. Where possible, "conventional" notation, as found in mathematical exposition and in other programming languages, is employed so that a reader's intuition will generally lead to a correct interpretation of SEMANOL(76) code. The semantics of execution are described by the use of SEMANOL(76) in terms of parse trees and elements of the original source program text, and so can be directly understood by the reader.

While the SEMANOL(76) metalanguage has many features common to other programming languages, its unique domain of application means that many of its features are not so common. Like many programming languages, it has: imperative, conditional, and repetition control statements; Boolean constants and functions; procedure definitions; recursion abilities; a rich set of character string operators; functional definitions that provide case selection; etc. However, many other features are unusual. SEMANOL(76) provides facilities for defining a context-free grammar, including a feature for context-sensitive specification of where spaces may appear, and couples that with an operator for generating a parse tree for a given string from that grammar. Various operators are then available for use upon this parse tree, including a group for tree traversal. SEMANOL(76) deals with sequences and offers high-level iterators, including existence tests, for use on these structures. Arithmetic is done on numeric strings and so has a significance that is independent of host machine factors; the arithmetic specified for Minimal BASIC is controlled by the one doing the specification in SEMANOL(76) and not dictated by the fact that the SEMANOL(76) system is implemented on a HIS-6180 computer. Assignment and reference are accomplished through use of a single level associative storage mechanism. An effort has been made in designing, and revising, the SEMANOL(76) metalanguage to provide these features in a manner that would stress readability; it is a language where the prospective reader's viewpoint has been a dominant influence.

Programs written in SEMANOL(76), such as the specification of Minimal BASIC, can then be processed by the SEMANOL(76) Interpreter. The SEMANOL(76) Interpreter accepts a SEMANOL(76) specification of a programming language and uses that input specification to realize the semantic effect of (i.e., to execute) programs written in the language thus defined. By virtue of the Interpreter, SEMANOL(76) specifications can themselves be tested and debugged. Furthermore, an operational standard for the defined language is thus created.

The operation of the elements that constitute the SEMANOL(76) system is shown in Figure 2. The broken line encloses the SEMANOL(76) Interpreter, which can be seen to actually consist of two loosely connected programs identified as the Translator and the Executer. The Translator accepts the SEMANOL(76) program describing a programming language and converts it to SIL code. The SIL code is an alphanumeric representation that is much more conveniently processed than the original text. The SIL file is read by the Executer program, and the SIL code is then used to control, or drive, the Executer program. The present Interpreter is operational upon the HIS-6180 Multics System.

It is to be emphasized that this sytem is interpretive, and that neither the defined language program, nor the SEMANOL(76) program describing the defined language, are translated (i.e., compiled) to machine code. The Minimal BASIC program text is interpretively "executed" by the SEMANOL(76) program describing the Minimal BASIC language, while the SEMANOL(76) program text (i.e., the SIL code) is, in turn, interpreted by the Executer program.

This two-level interpretation does mean that the execution time of test programs is slow. The fact that test programs are small has made the situation acceptable. However, as the SEMANOL(76) system has begun to emerge as a prospective part of language standardization and control, the importance of efficiency has increased. Thus efforts are underway to improve the current programs and also to develop a new design. The new design would move in the direction of changing the current Translator into a generator of executable code (i.e., it would generate interpreters). This new design would remove one level of interpretation and so provide a substantially more efficient process.

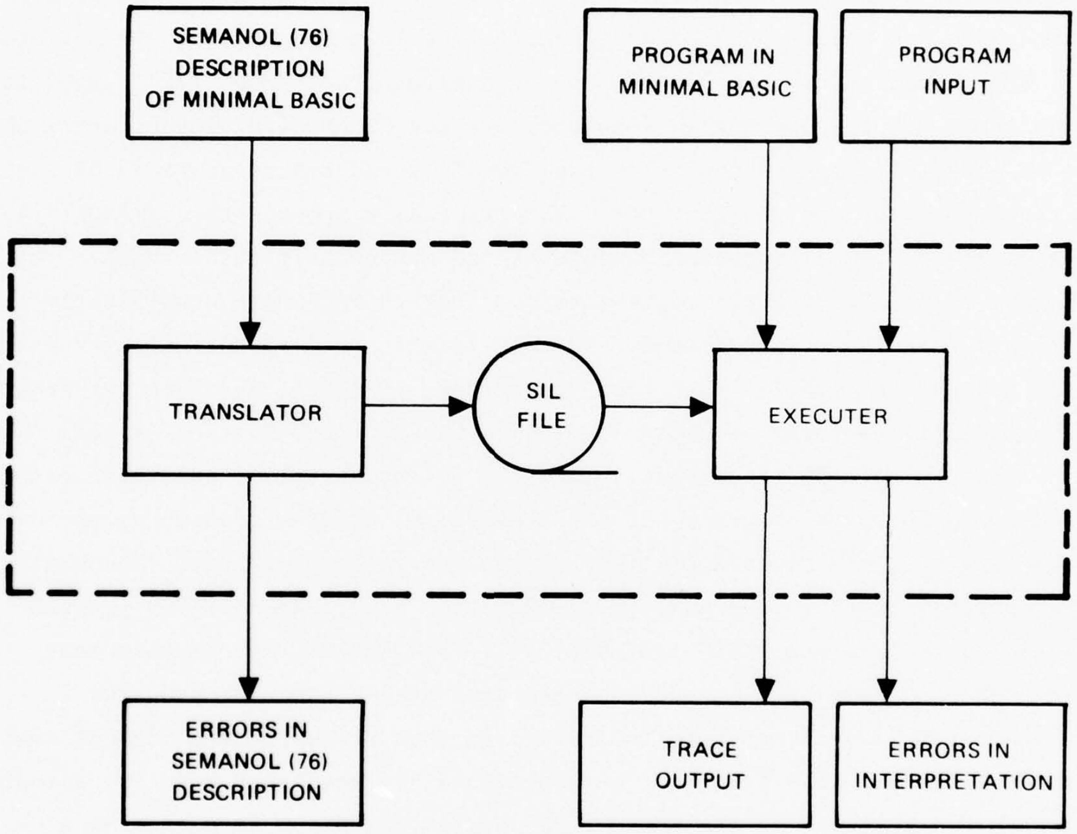


Figure 2: SEMANOL(76) Interpreter Logic

THE MINIMAL BASIC SPECIFICATION

The production of a SEMANOL(76) metalanguage specification of the Minimal BASIC programming language was the major accomplishment of this project. This specification is complete and was extensively computer tested; it is believed to be in sound condition. The nature of this specification of Minimal BASIC is discussed hereinafter.

The SEMANOL(76) metalanguage provides a normalized notation to be used for describing the syntax and semantics of programming languages. The metalanguage, and certainly the underlying theory of semantics as well as the specification conventions used with SEMANOL(76), result in a programming language being defined in operational, interpretive, terms. Indeed, it is fair to think of the SEMANOL(76) metalanguage as a programming language meant to be used in writing programs that are interpreters of source text strings of the language being defined. Since the Minimal BASIC specification is such a program, it is described here largely in its role as a computer program that processes Minimal BASIC program text. This means that the organization of the specification is dealt with, significant data structures discussed, modeling algorithms described, and stylistic conventions generally mentioned. As with any conventional document or computer program, the manner in which the Minimal BASIC specification was written reflects a certain individualism, even though constrained by conventions and the application of standardized techniques. Hence, the Minimal BASIC "specification program" presented here is certainly not the only one that could be allowed; nor is it likely to satisfy all readers with regard to its readability (despite our hopes that it might). But it has been carefully computer tested, and this does give us confidence that it is a valid specification of Minimal BASIC.

We should note that this section of the report is meant to serve as an introduction to the formal SEMANOL(76) specification of Minimal BASIC as well as to be a self-sufficient description of that specification. So that the text may do both, this section uses the actual names appearing in the semantic and syntactic definitions of the specification. These names are thought to be suggestive enough of their definitions that they may be informally used here without comment (i.e., they are strongly intuitive). The definition names are generally hyphenated, and can be recongized in that way (e.g., numeric-function, numeric-let-statement-effect).

The operation of the specification program is shown in Figure 3. The text of a candidate Minimal BASIC program is first parsed. This parse step uses a context-free grammar, given by SEMANOL(76) syntactic definitions, and is accomplished in response to the #CONTEXT-FREE-PARSE operator of SEMANOL(76). A successful parse is followed by testing of the parsed program to determine that it complies with the seventeen context-sensitive restraints that valid Minimal BASIC programs must meet. Syntactically valid programs then have their execution effects (i.e., semantics) precisely modeled in terms of program operation. Program operation is modeled with regard to statement effects, statement sequencing, expression evaluation, and storage modeling. Because this specification was to be tested and otherwise usable, it is a specification of a specific language processor; it thus contains semantic definitions that explain execution in very exact terms for a specific computer. This general approach is expanded upon in what follows.

The first element of the SEMANOL(76) specification program is a declarations section; this section contains the names of any global variables used in the specification of Minimal BASIC and the names of syntactic components. Only 12 global variables are used in describing Minimal BASIC, and the use of most of these is described subsequently. Syntactic components, on the other hand, are declared in generous number. Because SEMANOL(76) is a functional language, there are many functions in the Minimal BASIC specification that are invoked repeatedly with constant parse tree arguments. These functions would thus compute invariant values needlessly. By declaring the names of those definitions that produce constant values for a given parse tree argument, the SEMANOL(76) Interpreter program is told to perform the computation of these functions only once for a given argument. The computed value is then associated with the argument node on the parse tree; later invocations of the function for that node need only retrieve the saved value rather than perform the computation. Syntactic components thereby contribute to processing efficiency but not to semantic description per se. Their use does mean that there is little performance penalty paid for being able to widely employ a readable functional notation.

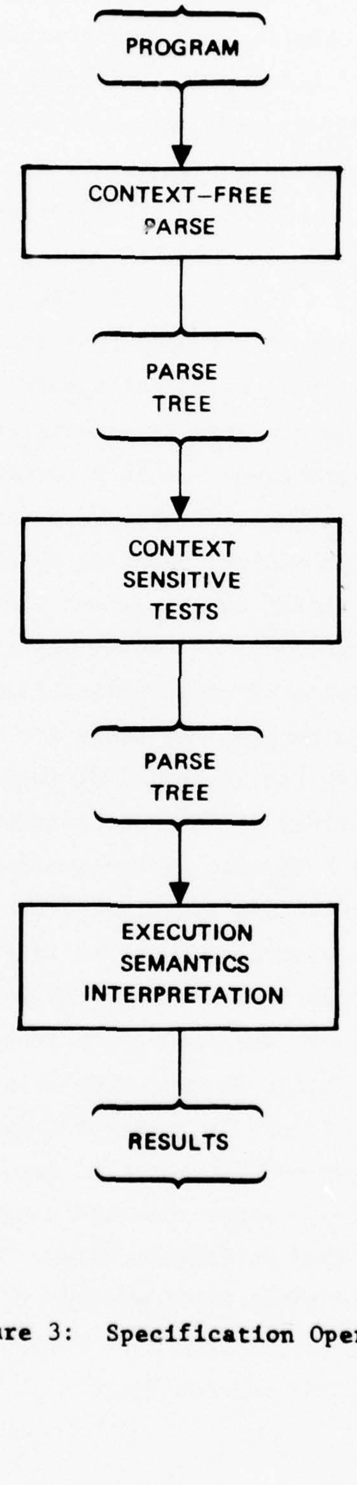


Figure 3: **Specification Operation**

The first operational step involved in defining Minimal BASIC is to parse the given program text. The syntactic definition structure used in the SEMANOL(76) metalanguage is similar to other context-free syntax notations. Thus the grammar we have defined for Minimal BASIC is able to be similar to that of the X3J2 document; it differs only when necessary to support semantic description or to correct discovered ambiguities. The result of parsing an input string with the grammar is to transform the given source program into a corresponding parse tree representation. If the program cannot be parsed or would yield multiple parses, a diagnostic message results. Observe that the second error condition reveals that the grammar defined by the SEMANOL(76) definitions is ambiguous; hence, the formal specification is itself in error. Since the SEMANOL(76) parsing operator is unrestrictive, the grammars that are allowed cannot be proven unambiguous but only carefully tested in an effort to avoid ambiguity. The specification thus recognizes this potential residual error condition. (We have considered changing SEMANOL(76) in this regard, so that grammars would be restricted to the extent that they could be proven unambiguous, but have not yet convinced ourselves that this is appropriate for a formal description system of wide applicability). The parse tree representation formed here then becomes the basis for further specification.

Following a successful parse, a series of tests is made upon the parse tree representation to decide whether or not the context-sensitive syntactic restrictions of Minimal BASIC have been met. These syntactic restrictions are conditions that cannot be expressed in a context-free grammar, and so must be separately prescribed. The restrictions given in the SEMANOL(76) specification for Minimal BASIC are drawn directly from the X3J2 document, so their inclusion needs no further justfication. The imposition of these restrictions at this point in the specification, prior to execution (e.g., as a compiler implementation would commonly impose them), is in keeping with the X3J2 document. For example, the requirement that numeric-function names be unique is meant to apply to a program as a whole, rather than applying only to numeric-function names that are actually invoked during execution. Observe that the notion of a legal (i.e., conforming) Minimal BASIC program is affected by the choice of when the test is applied; thus the placement chosen for these tests within the Minimal BASIC specification is significant.

The specifications of these context-sensitive tests use the high-level iterators of SEMANOL(76), #FOR-ALL and #THERE-EXISTS, especially, to advantage. The most prevalent approach is to construct a sequence of program elements of interest, and then to test each in turn against the remainder of the program for existence, or lack, of the desired condition. For example, a #SEQUENCE-OF numeric-function-reference nodes is formed from the parse tree in one test. (The elements of this sequence are easily specified since the parsing process has caused each such reference to be identified and the associated syntax class name to be included in the parse-tree representation.) Each reference is selected in turn by the #FOR-ALL iterator, and tested against all the elements of the #SEQUENCE-OF def-statements to see if a name match exists. If not, a "missing-function-definition" condition has been found; the prospective Minimal BASIC program is thereby deemed illegal, and it is interpreted further only to the extent that the remaining context-sensitive tests are performed. Note that the def-statement sequence is declared as a syntactic component, and so is computed only once in this process (even though not explicitly computed and saved). The iterator and sequencing operators of SEMANOL(76), combined with the use of parsed representation, result in these tests being stated succinctly and clearly.

The semantics of program execution are then explained in operational terms. A top-level control section is used to describe both the computational effects of executing a statement and the execution successor for each statement. The computational consequences are described in a series of semantic effects definitions, a set of definitions being associated with each statement class and thoroughly treating all possibilities. Similarly, sequencing rules are given by a series of successor definitions. Major subsidiary parts of the specification then deal with evaluation details and standard names used in storage modeling. The various Minimal BASIC statement types are considered within this framework in what follows.

The primary computational elements in our specification of Minimal BASIC are the numeric-let-statement, string-let-statement, input-statement, read-statement, and print-statement; the consequence of executing each is given in an associated "effect" semantic definition (e.g., numeric-let-statement-effect).

The numeric-let-statement-effect is to assign the value of the right-hand-side expression to the left-hand-side name. The semantics of evaluation are then contained in numeric-value, described later, while the formation of the left-hand-side name is described by a standard-name definition, also described later. Thus the high level semantics are given in this control section, with the details to be found in subsidiary semantic definitions. The semantics of the string-let statement are handled in an analogous manner, in this case using string-value for right-hand-side evaluation.

The effect of input-statement execution is given for the interactive case. This means that an input-prompt-character is first output. After that, an input-reply is read, checked, and its values assigned. The output listing produced by operation of this definition simulates that which would be produced at a terminal during an interactive input dialogue. The semantic definition here makes use of the small context-free grammar for input-reply to parse the input string for ease of processing; it also uses the grammar of numeric-constant in a recognition test. The conversion of numeric input strings for assignment is accomplished by the implementation-numeric-representation conversion operator (considered later), thus defining the mapping from an external representation to an internal representation (the X3J2 document expects such a distinction, in recognition of the use of various computer dependent arithmetics). No such mapping is needed for strings,eexcept possibly to remove quotes, since the semantics of string operations are independent of implementation representation. Much of the semantics of the input-statement then deal with testing the input-reply for correctness (e.g., the right number of data items, matching types, items being in range) and describing the action to be taken in the case of error. All errors result in diagnostic messages being sent to the user and the input-statement processing being repeated; that is, the semantics call for the user to re-submit incorrect input-replies. Note that no assignments are made until all data items in an input-reply have been judged valid.

The series of input-replies for a Minimal BASIC program is read as a single block of characters by the #INPUT operator, when processing the first execution of an input statement, and assigned to the global variable input-file. The modeling of input-statement effects then provides for unblocking these input-replies. This is accomplished by scanning to the end-of-input-reply-char, assigning the input-reply thus delimited to the global variable input-line, and then removing this input-reply from the input-file string. The model thus does not literally provide interaction, but does permit any input dialogue to be simulated.

The semantics of the read-statement are very similar, and the conversions of strings to internal representations are identical with those of the input-statement. The error semantics, naturally, differ somewhat. The current position in the sequence of data items, extracted from data-statements, is maintained in the global variable data-list-pointer. This position index is incremented each time an element is read, and is reset to one by the restore-statement-effect. It should be noted that the conversion of data-statement constants must be specified to (effectively) occur at read-statement invocation time, as described here, since the type of the data items cannot be determined by their syntax; the type of the constants depends upon the type of the variables to which they are assigned. This condition is certainly implied by the SEMANOL(76) specification, but it may be a case in which the constraints upon an implementation are not as obvious as one would like.

The semantics of print-statement-effect are lengthy because the formatting possibilities of Minimal BASIC are rather extensive; however, they are generally straightforward except for numeric conversion specification. Numeric conversion is complicated because the printed form depends upon the value of a number, different ranges having different formats, and because implementation factors intrude here. The approach used is to convert the implementation-number representation to the corresponding canonical-form, a string format that conforms to a reduced Minimal BASIC syntax, and then to apply the numeric-output-representation operator to this canonical-form. The translation to canonical-form is implementation dependent; however, the numeric-output-representation semantics are not dependent on internal numeric representation factors. Numeric-output-representation classifies the canonical-form, selects the printing format and converts the canonical-form to the format selected. The implementation dependent factors that influence print representation (e.g., significance width, class ranges) are applied in this process. It is important to realize that the implementation parameters of internal value representation and those of output print representation, while naturally related in implementations, are given separately by this method; a great deal of flexibility is thereby provided with which to describe implementation families.

Complementing the description of statement effects is the specification of statement sequencing semantics. The same simple sequencing rule applies to all the statements just described; namely, advance to the next executable statement. This is described in terms of the global variable current-statement and the sequence-of-statements-in the Minimal BASIC program. The sequence-of-statements-in definition directly forms an ordered list of statement nodes from the parse tree representation, while current-statement then contains the node of the statement presently being interpreted. Simple sequencing, as done here, then involves moving forward from the current-statement position along the sequence-of-statements-in the program to the next executable statement node. This successor node is then assigned to current-statement, and interpreted of that statement initiated. The semantics of simple-control-statements are found in the specification of their successor semantics; they have no effective computational part. For instance, the branching influence of a goto-statement is found in the definition of goto-statement-successor. Here, the destination line number is extracted and a statement line with a matching line number value is sought. If one is found, the node of that statement is assigned to current-statement and the control branch thus realized. The situation for if-then-statements and on-goto-statements is similar.

However, other control statements are more difficult to model. The gosub-statement has an effects part that locates the node of the first executable statement after the gosub-statement and places this at the front of a sequence assigned to the global variable return-point-list. A semantic test is made at this step to see that the length of this sequence, which corresponds to the number of outstanding gosub-statements, is within its established limit. The successor of a gosub-statement is then found exactly as was the goto-statement successor. The semantics of the accompanying return-statement are then given by return-statement-successor. That specification calls for the statement node at the head of the return-point-list to become the current-statement, after which the node is removed from the return-point-list. By this means, a simple stack model is used to describe gosub/return semantics.

The control semantics of for-blocks are more complicated. They are essentially described in terms of a sequence of triples, the sequence being assigned to the global variable active-for-block list. Each triple consists of the control variable, the value of the limit, and the value of the increment associated with a particular for-block. This entry is made when the effect of a for-statement is being computed upon for-block initialization. At the same time, the initial value of the loop variable is found and assigned to the control variable. The successor of a for-statement depends upon the value of the control variable. If the limit value has not been exceeded, then the next executable-statement (i.e., the first statement in the for-block) is made the current-statement. If the limit has been exceeded, the most immediate lexically following next-statement having the same control variable is sought, and its successor becomes the current-statement. Additionally, the triple for the terminated for-statement is deleted from the active-for-block-list. This specification makes clear the nature of loop testing and emphasizes the static notion of for-blocks that has been adopted for Minimal BASIC. The effect semantics of the next-statement cause the value of the control variable to be modified by the loop increment value held in the active-for-block-list sequence. The successor of a next-statement is always the most immediately prior for-statement with the same control variable (another affirmation of the static for-block structure). Note that a global variable is used in our specification to distinguish between block initialization and iteration. This variable, first-time-through, is set #FALSE when iteration via next-statement successor is occurring; that setting prevents for-statement creation of another active-for-block-entry for that for-statement, and it then is immediately set #TRUE. This variable is thus #FALSE only for the very brief interval between the completion of next-statement interpretation and initial testing within the definition of for-statement-effect.

The semantics of storage modeling, for assignment and reference, are provided by the facilities of SEMANOL(76) and the definition standard-name-of. The SEMANOL(76) functions of #ASSIGN-LATEST-VALUE and #LATEST-VALUE are used directly throughout the specification, with the naming reference being formed by standard-name-of. Because Minimal BASIC has only very simple data structures and referencing rules, there is no need for complex storage models; hence, abstract reference names can be used without any need to descend to bit-level machine parameterization. Simple-numeric-variables and string-variables retain the name given in the Minimal BASIC text (e.g., "N" has a standard-name-of "N", "C$" remains "C$"). Parameters in function definitions are qualified by the function-name (e.g., "FNK(A)" would lead to a standard-name-of "FNKA"); this simple device satisfies the scoping rules of Minimal BASIC. Numeric-array-elements follow much the same pattern (e.g., "B(4,2)" becomes "B(4,2)" in this process) but are somewhat more complex to describe since subscript expressions must be evaluated and the value then normalized. This normalization causes subscript values to be rounded to integers, converted to canonical-form, and have any leading zeros removed. For instance, "01", "1", and "11E-1" are all translated to "1". Subscript values are also tested against their allowed range, whether declared or implied; thus this semantic specification provides run-time bounds checking. This method of name derivation is simple and direct; it should be easily understood.

Expression evaluation semantics are given by the string-value and numeric-value definitions. Since Minimal BASIC does not have string operators, string-value is virtually trivial and need only recall a string-variable from storage or remove quotes from a string-constant. Note again that the internal representation of a string is identical to its representation in the Minimal BASIC programming language (and that this representation is likewise identical with that of SEMANOL(76)); thus the semantics are indeed implementation independent. Numeric-value is more complicated since Minimal BASIC provides a full range of numeric expression possibilities. Numeric-value definition also requires that the details of some implementation arithmetic be included. That is, a SEMANOL(76) specification is always an implementation if it is to be complete and so able to produce interpretive results; therefore, details of a precise arithmetic (as well as data representations) must be given. It is suggested later how a specification might be written to define implementation constraints in terms of "closeness" to standard results. While that method was not entirely implemented in our specification, our specification was written in anticipation that the method might be used and this consideration will be evident in the specification.

Numeric evaluation thus uses three forms of arithmetic: implementation arithmetic, standard arithmetic, and canonical arithmetic. Operators for each class are then identified by using one of these three terms as a prefix (e.g., standard-add, implementation-add). Standard arithmetic is a high precision arithmetic that is implementation independent; it is used to define standard results against which implementation results might be tested. Canonical arithmetic is a subsidiary arithmetic, also implementation independent, that is primarily used in defining standard arithmetic. Standard arithmetic operates upon numeric-constants of any allowed Minimal BASIC syntax, while canonical arithmetic operates upon constants of a more uniform syntax. By converting constants to canonical form, and performing arithmetic operations upon canonical constants, the semantics of evaluation are thought to be more easily explained. The result of applying standard operators is to generate canonical constants (since some choice of result representation must be made); canonical operations likewise yield canonical results. One could, of course, write the Minimal BASIC specification so that the standard arithmetic were used as the implementation arithmetic; we have essentially done this in past specifications. However, we have introduced a distinct implementation arithmetic so that constraints might later be included more easily. This approach may also help make clearer where implementation defined factors intrude upon the programming language definition, since each implementation specification can only be made complete by providing descriptions for these definition names.

Since we have no particular computer to model, the implementation arithmetic specified is a decimal one that is intended to provide the minimum significance and range called for in the X3J2 draft proposal. This implementation arithmetic is able to use the definitions of standard arithmetic, and so it is briefly described in our specification. Various implementation factors are then parameterized, examples being implementation-precision, implementation-infinity, and implementation-zero. A variety of decimal implementations can thereby be easily formulated. Implementation operations are tested for causing overflow or underflow, attempting division by zero, or attempting an undefined form of involution. The response to these actions is defined to be implementation independent in the draft proposal, and so is part of the evaluation semantics, although the specific result is itself implementation determined.

For instance, implementation underflow is always to be responded to by returning a value of implementation-zero; the algorithm thus appears in an implementation independent part of the Minimal BASIC specification, but the definition of zero is part of the implementation dependent semantics section.

The semantics of numeric-value are then given in a recursive way (e.g., the numeric-value of "A+B" is given by applying the implementation-add operator to the numeric-values of the two operands), with the rescursion being grounded in constants, variable references, or function calls. Numeric constants are translated to an implementation representation by exactly the same method as are input constants. The values associated with variable references are in an implementation form already, thus need only be recalled by #LATEST-VALUE. Function calls are processed by invoking a semantic definition that (1) determines the numeric-value of any argument-expression, (2) assigns this value to the standard-name of the dummy parameter, and (3) then causes the numeric-value of the function expression to be determined. Function definitions are very simple in Minimal BASIC, thus their semantic description is likewise uncomplicated by obscure control factors.

A portion of the semantic description of evaluation, named relation-value, deals with specifying the nature of relational tests. The general structure here is much like that used with expression evaluation in that there is a string-relation-value part and a numeric-relation-value part. The string-relation-value part deals only with equality or non-equality, and it deals directly with the string representations since their syntax is identical to that of SEMANOL(76) strings. The numeric-relation-value part must consider a variety of relational test options and do so with regard to the implementation sensitive nature of these tests. The semantics of each test are thus given in implementation terms upon implementation numbers. So, just as there is an implementation arithmetic, there is a set of implementation relational operators (e.g., implementation-equals-test). These implementation relational operators, as with arithmetic, are given in our specification in terms of the corresponding standard relational operators.

The evaluation of numeric-supplied-functions (e.g., SIN, COS, LOG) is intended to be implementation specific in Minimal BASIC. In our specification, this is shown by providing, in the implementation dependent part of the specification, a definition for each function that is an interface to a corresponding external procedure. SEMANOL(76) has an #EXTERNAL-CALL-OF operator for such linking; the interface definitions are then responsible for mapping the internal representation of the actual argument(s) to the string format needed by the external procedure, and then mapping the result back to the internal format needed. By this device, library procedures can be used provided the library procedures themselves conform to the conventions of EXTERNAL-CALL-OF. In particular, the external procedures must accept and return string values.

While this description of evaluation is brief, the code involved is rather lengthy because of a need to consider the boundary conditions and otherwise provide an implementation model of semantics, something other formal description methods do not generally offer. The presentation is meant to be one that can be read down to the level of desired detail; thus arithmetic intricacies can be avoided if unwanted.

In addition to the specification elements already discussed, there is a part of the specification that is composed of a collection of definitions that constitute selector functions. These selector functions are applied to the parse tree to return specified nodes or values and to test the syntactic classification of a given node. These functions are meant to be obviously named in order that overall specification readability can be increased by reference to them. They have been collected together because of their wide applicability and commonality of purpose; they provide a low level of detail and are correspondingly simple in their operation.

While a few forms of error testing were mentioned, the specification throughout attempts to detect any "run-time" error that can occur. In general, these were recognized in the X3J2 document and the response provided in the specification agrees with that report. Since SEMANOL(76) provides an interpreter approach to specification, the description of these error conditions and response semantics is easily done. The specification also has been written so that each semantic definition includes, as a comment, an assertion about the nature of its input values. The reader can thus readily ascertain what form of arguments each definition is expected to receive. The assertions are expressed in SEMANOL(76) notation so that their meaning is precisely given, but their correctness has not been computer tested. (To confirm these assertions operationally is presently awkward, and would lengthen the specification to no great advantage. We are contemplating the inclusion of an operational assertion testing facility in later systems.)

QUESTIONS OF INTERPRETATION

The preparation of the formal SEMANOL(76) specification of Minimal BASIC necessarily demanded a careful reading of the X3J2 document upon which the formal specification was to be based. This analysis was done so that the needed details of Minimal BASIC could be determined and a complete, precise model of the Minimal BASIC programming language thereby formulated. Since a SEMANOL(76) specification is a type of program, this process is similar to that involved in translating any prose description into a computer program; as is true generally, this process of "implementation" revealed many cases in which the X3J2 prose document was unclear or incomplete. This was true despite the great efforts made by the TRW project group to resolve such questions through continuing analysis of the X3J2 document; that is, we made a much more intensive effort to resolve problems than we would expect most other readers of the draft proposal to make.

Our analysis was also unique in that we deliberately sought to identify statements that seemed to allow multiple interpretations. We were not seeking to write a formal specification that could be defended as consistent with some interpretation of the X3J2 document, as one might do when implementing a language processor, rather we were attempting to formalize (i.e., make more precise) the notion of Minimal BASIC given in that X3J2 document. Thus we sought to be very sensitive about recognizing the points at which we were making decisions that were at all questionable with regard to reasonable interpretations of the draft proposal. While this can sometimes seem to be quibbling, it is important that an ANSI standard be uniformly understood; in our case, it is also important that the derivation of the SEMANOL(76) specification be explained.

The list of items given here reflects problems in the X3J2/76-01 draft, each of which was reported to RADC in considerably greater detail. Each item is then accompanied by the interpretation that was provided in the SEMANOL(76) specification that was delivered. These items reflect one consequence of using the formal SEMANOL(76) methods; namely, the identification of areas of imprecision and incompleteness in conventional programming language specifications. They also reveal problems to which we had to devote considerable energy. The ambiguities were these:

1. While the X3J2 draft proposal does consider implementation representation of numbers, unlike many such specifications, its coverage is inadequate. It fails to define the sign of zero, if zero is uniquely represented, or to say if both plus and minus zero are to be allowed; contemporary computers are certainly not uniform in this regard, so different readers may naturally come to different conclusions. The answer to this question can affect (1) the value of relational expressions and (2) the value of zero divided by zero, since the result of division by zero is to have a magnitude of machine infinity and the sign of the (zero) numerator. There is also a question as to whether the implication that the representations of the largest magnitude values (i.e., machine infinities) and of the smallest non-zero magnitude values (i.e., machine infinitesimals) are to be the same for the corresponding positive and negative cases is intended. That is, a symmetric representation seems to be intended (e.g., negative-machine-infinity and positive-machine-infinity will have equal magnitudes). While this inference seems reasonably clear, and the benefits of symmetric representation are recognized, many current machines do not support such a symmetry, and to impose this requirement for them may lead to unwanted inefficiency in conforming implementations. Because of this, a question of intended meaning does exist.

To provide generality, our specification includes both positive-machine-infinity and negative-machine-infinity so that these magnitudes may be defined independently. A positive-machine-infinitesimal and negative-machine-infinitesimal are specified in an analogous way. For clarity and semantic predictability, we have defined zero to have a single representation, machine-zero. These parameters are then used in the implementation dependent semantics so that each representable number cannot be greater than positive-machine-infinity nor less than negative-machine-infinity. Furthermore, only one of the relations of equality, greater than, or less than can hold between two numeric representations. The only numeric representation between positive-machine-infinitesimal and negative-machine-infinitesimal is machine-zero. This choice of implementation parameters should allow a full range of Minimal BASIC language processors to be modeled precisely. The specification delivered has defined the magnitudes of non-zero machine limits to be the same in both positive and negative directions; it thus agrees with the draft proposal implications. The sign of zero has been specified to be plus, as that choice seems "natural".

2. The description of for-blocks is especially troublesome since the draft proposal, beyond being unclear, did not permit us to derive a defensible model from what was given. Much of this problem appears to result from inadequately distinguishing between the dynamic and static properties of for-blocks. While it is recognized that both compiler and interpreter oriented processors are to be allowed, the wording of the draft proposal does not seem to be consistent with any implementation model.

   We have chosen to prepare a specification that seems compatible with what we believe the draft standard is attempting to say. In this specification, the for-statement and the next-statement are defined in conjunction with each other. The physical sequence of statements beginning with a for-statement and continuing up to and including the first next-statement with the same control variable is a "for-block". The first and last statements in a for-block are its for-statement and its next-statement, respectively. Any other statements in a for-block are its interior statements.

   Each minimal BASIC for-block is either inactive or active; each is inactive at the initiation of a program. A for-block becomes active only upon execution of its for-statement; thus a physically contained for-block does not become active upon the execution of the for-statement of one of its physically containing for-blocks. An active for-block becomes inactive only when either (1) the normal exit from the for-block is taken or (2) control is transferred to a for-statement (which may or may not be the one associated with that for-block) having the same control-variable. Thus contained for-blocks are not deactivated when containing ones are.

   A program attempt to execute the next-statement of an inactive for-block is an error.

3. As given, the semantic definition of the draft proposal can generally be construed to define a static declarative nature for DIM and OPTION statements that is unaffected by whether such statements are executed or not (i.e., such statements are always declarative and may also be executed). However, this matter is confused by the resolved questions part of the draft proposal which fails to adequately recognize the differences between static and dynamic interpretations that can exist. In particular, it is not made clear how initial (pre-execution) conditions are established; this is especially needed so that later enhanced versions that permit (multiple) executable declarations may be consistent with this minimal version.

   Our specification treats DIM and OPTION statements as being purely declarative; they are not executable-statements. Only one OPTION statement is allowed in a program, and only a single dimension-statement may refer to a given array name; these rules are imposed by context sensitive restrictions and so enforce a static interpretation of these statements.

4. The number of printable characters in an implementation output line, called the "margin", is not clearly defined nor is its execution effect. In the definition of margin, it is unclear whether the end-of-print-line character (or characters) is to be included or not in the the margin count. The description of print then is contradictory as to whether the last print position of an output line can be used, the confusion being introduced for print items that exactly fill out the last print zone of a line and so cause the columnar position, which points to the next available print position, to exceed the margin. The draft proposal seems to say that such items are to be printed on the next line.

   Our specification does not consider the end-of-print-line token to be included in the margin value. It also assumes that the last print position in a line is meant to be used for printable characters; thus it will allow print items to exactly fill the last print zone of a line.

5. There were two unresolved questions relative to NR2 print conversion. First, if d (the number of significant decimal digits to print) is given an implementation value less than that required to exactly represent every implementation numeric representation, then it is not stated whether truncation or rounding should be used for reducing the precision to d significant digits. Second, the criterion given for selecting NR2 form conversion is unclear in defining a decimal interval and is inappropriate for non-decimal implementations.

   Our specification first translates an implementation numeric representation to a standardized output numeric representation. While the mapping itself is implementation specific, the numeric output representation, which conforms to the syntax of Minimal BASIC, is not. Decimal rounding is then applied to this output representation when the precision of a constant must be reduced for printing. The test for NR2 form conversion is applied after rounding and employs the inclusive interval [.1,99:::9], where d 9's are used in the upper bound. This interval is consistent with the draft proposal if decimal rounding is applied to a decimal numeric representation. (But observe that if an implementation numeric representation is not decimal, specifying rounding and the NR2 interval in decimal terms can impose unexpected processing burdens on an implementation.)

6. It is unclear whether leading zeros in the exrad of NR3 conversion forms for printing are permitted.

   Our specification does not provide leading zeros.

7. The effect of the comma separator in print lines is ambiguous since the phrase "current print zone", used in describing that effect, is nowhere defined. Specifically, the evaluation of the comma when the columnar position is equal to the first print position of a zone is uncertain; should it have no effect or generate a zone's worth of spaces?

   Our specification effectively defines the current print zone to be the one that includes the present columnar position (rather than the print zone that received the last output character). Thus a zone of spaces, or an end-of-print-line, is generated in the situation just described. This specification agrees with a comment made in the resolved questions part of the draft proposal.

8. The issue of when a print line must be presented to the user is unanswered in the draft proposal. Is generation of an end-of-print-line sufficient to cause the current print line to be immediately output? What happens if the current print line is not empty when program execution terminates?

   Our specification assumes that end-of-print-line generation is necessary and sufficient to cause the current line to be transferred to the output device; on termination, a non-empty current line is returned to the user. In large measure, this solution mimics that of conventional interactive implementations.

9. The specification of what is to be done upon encountering underflow when converting a number in an input-reply is directly contradictory in the draft proposal; one statement is made that the value underflowing should be replaced by zero, while elsewhere it is stated that such underflow is an error.

   The weight of the draft proposal seems to be on the side of assigning a zero value, thus that is what we have specified in all cases. Additionally, our specification is written so that generation of a non-fatal error message is made an implementation option that is determined by a parameter setting. Thus production of this message is easily turned on or off.

10. It is not specifically stated whether or not program constant conversion must agree with the conversion applied to constants received by INPUT; that is, whether the implementation representations of two syntactically identical constants, one of which appears in the Minimal BASIC program itself and the other of which appears in an input reply, are to be identical.

    While nothing specific is said, the tone and organization of the specification imply that the conversions should be uniform. We use the same implementation conversion rule in both cases, and so explicitly show that these conversions are meant to be the same.

11. The action to be taken in some error conditions is unspecified; these are instances in which a given property "must" (or "shall") obtain but which is ignored in the error conditions section part of the draft proposal. Specifically,

    (1) "The value of the integer represented by a line-number must be non-zero;..."

    (2) "The declaration for an array, if present at all, must occur in a lowered numbered line than any reference to an element of that array."

    (3) "...(the number of arguments must correspond exactly to the number of parameters)..."

    (4) "A function definition shall occur in a lower numbered line than that of the first reference to the function"

    We have taken each of these to be a fatal error that is detected by a context-sensitive test applied prior to execution interpretation.

12. The rules that govern the appearance and treatment of strings having a length exceeding 18 characters seem unclear. Strings may appear in

    (1) relational-expressions within if-then-statements,
    (2) string-expressions in string-let-statements,
    (3) string-expressions in print-items,
    (4) input-replies and data-statements.

    The treatment to be given long strings in each of these cases is not given explicitly, or seems inconsistent in some instances.

    Our specification imposes no syntactic restrictions upon string length in either the context-free grammar or in the context sensitive restrictions initially applied. Thus long strings may appear freely within the text of a Minimal BASIC program. String expression evaluation is done dynamically, but it too disregards string length. However, assignment to string variables is subject to the condition that strings have a length equal to or less than max-assignable-string-length, an implementation parameter whose integer value must be at least 18. This specification permits long strings to be printed, as clearly intended in the draft proposal, and also to be tested in relational-expressions. It also permits long string constants to appear in string-let-statements, but such assignments are prevented (i.e., an attempt to execute such a statement is an error). Similarly, long string constants may appear in data-statements but cannot be converted when processed by a read-statement. Long strings in input-replies are likewise rejected.

13. There is a question as to whether it is legal to have parameterized user-defined-function definitions that do not reference the parameter in the expression. For example, is

```
100  DEF  FNZ(X)=A(9)*4
```

to be allowed?

Our specification allows such usage; thus the example is considered legal.

There are a number of lesser ambiguities that provoked considerable discussion and analysis of the draft proposal. In these cases, a "reasonable" interpretation could be constructed without great difficulty; however, these interpretations carried with them a recognized degree of doubt that others would arrive at these same conclusions. These ambiguities are examples of troublesome wording that should be clarified to improve the uniformity of reader understanding of the draft proposal. (We have suggested editorial changes for most of these in other reports.) Because of the critical analysis that we were attempting to do, these ambiguities probably caused us more trouble than they would other readers. These ambiguities include the following:

1. The nature of user function definition, whether by def-statement declaration or execution, is uncertain, largely because of confusion elsewhere about the dynamic and static qualities of other declarative statements. Our specification considers user function definition to be a static language property; def-statements are not treated as executable statements.

2. Except in the formal syntax, there is a general failure to distinguish between "statements" and "lines". The term "statement line" is sometimes used in an attempt to gain greater clarity. The intended meaning can be determined, but the critical reader's task is more difficult than it need be.

3. The phrase "line number" is sometimes used when "line number value" is indubitably meant.

4. The discussion of optimization is not clear at all. This is a problem that was reported, but that did not directly affect the specification written since the impact of optimization was ultimately ignored in that specification.

5. The description of gosub-statements is ambiguous in that one interpretation of the given wording would effectively allow an unlimited number of unreturned gosubs. Our specification introduces an implementation parameter for this number and assigns it a value of ten; this value is one interpretation, and a likely one, for the description that appears in the draft proposal.

FORMALIZATION OF IMPLEMENTATION DEPENDENCIES

Acceptable methods of characterizing those aspects of programming language design that are conventionally left to be defined through implementation are not yet available. The difficulty appears when attempting to formulate specifications so that:

1. Intentional machine and environmental differences may be clearly and precisely stated. At the very least, differences are expected to exist among language implementations because of differing data representations, computational incompatibilities, conflicting error treatments, operating system induced discrepancies, etc. The extent of differences among implementations will largely be determined by how strongly code transferability has influenced the design of a particular programming language, but differences are expected even for languages commonly considered "machine independent."

2. Optimization options can be formally given. Different language processors are expected to order operations differently and otherwise generate code that is meant to capitalize upon the unique features of a given machine. The code produced may also be influenced by the compiling techniques that are employed to improve compiler efficiency. Computing efficiency is thereby realized but in ways that are hard to describe precisely.

Thus one goal in writing language standards is the construction of a specification that describes the idealized (machine independent part of the) language, while also permitting clear identification and specification of features whose meanings are left to be the consequences of a given execution environment. If possible, the implementation dependent parts should be described to the extent that the range of acceptable implementations can be clearly made part of the language defining specification.

Because SEMANOL(76) is meant to provide an operational specification, the SEMANOL(76) specification must be made complete through some form of precise specification for all the machine and implementation factors that could otherwise be dismissed in a conventional specification method; machine details and evaluation order cannot be ignored here. That is, the complete SEMANOL(76) specification is itself an implementation, and consequently it must include a specification of semantically significant machine features. This is possible with SEMANOL(76) since its data representations and operators are indeed machine independent and capable of modeling conventional machine features. A conscientious effort is then made, as in writing the SEMANOL(76) specification of Minimal BASIC, to separate these machine details from the other code so that they do not obscure the broader semantic specification. For example, the implementation specific semantics of arithmetic operations are described by the SEMANOL(76) metalanguage, and then appear in a distinguished part of the specification that is meant to describe the machine-dependent elements of the semantics. Standards for similar computers can then be prepared by revising the SEMANOL(76) code that expresses these machine dependencies. In fact, a strong effort is made to parameterize these features so that a family of specifications can be built, each differing from the other only in the values given to these machine dependent parameters.

It should also be noted that machine details can oftentimes be given in external functions. The SEMANOL(76) metalanguage supports this option through the #EXTERNAL-CALL-OF feature. As a result, a library can be built that contains routines to simulate the hardware and operating system functions that are needed to complete the description of the language being defined. Implementation dialects can then be distinguished by separate libraries.

The difficulty with this type of approach is that it produces a specification particular to a given computer or, at best, to a class of similar machines. If the language under definition is to be widely used, the SEMANOL(76) method is then guilty of overspecification in some respects. (But note that some programming languages have been designed so that the areas of specification left for implementation determination are so extraordinarily broad that each language implementation can reflect virtually a different language. In such cases, any usefully complete specification method will necessarily correspond to an implementation and so might be thought overspecific.) For some language features, there may be no way to relieve this problem. In such cases, as for unrestricted pointer variables, it may simply be necessary to supply alternate definitions for each implementation that is envisioned. That is, there are likely to be programming language features for which suitable constraints cannot be formulated nor a suitable degree of abstraction obtained. However, for arithmetic it appears that SEMANOL(76) can provide a useful implementation independent way of characterizing what is to be allowed and doing so in a formal manner. This method was investigated in this contract relative to its use with Minimal BASIC and, although not yet fully developed, it appears very attractive.

The central idea is that (1) the representation of constants and the results of arithmetic operations can be given a "standard" definition, (2) each implementation will likewise be described in terms of its corresponding manner of representation and operation, and (3) the implementation and standard results must agree to some given precision. Thus a condition such as

standard-floating-add(x,y)-implementation-floating-add(x,y)$|<\varepsilon$(add,x,y)

must hold. The standard-floating-add function would be written in the SEMANOL(76) metalanguage since, as the standard function, it must be fixed as well as available for public review. We presently expect the standard arithmetic representations and operations to essentially reflect a machine whose precision exceeds that of contemporary machines. Implementation-floating-add can either be expressed in the SEMANOL(76) metalanguage or as an external function (but it must correspond to what happens for a given language processor upon a given host machine). The allowable difference, denoted by ε, would be given by SEMANOL(76) specification and could be expressed as a constant or a function. It would be desirable if ε might itself be expressed as an implementation independent constraint and so might be a fixed part of the standard definition; however, it is not clear that this can generally be done in an acceptable way. But even if ε is itself selected on an implementation-by-implementation basis, its appearance in the SEMANOL(76) specification provides a basis of conformance testing. It also makes clear the quality of a given implementation relative to a standard implementation, and can allow qualitative comparison of various implementations to be done.

This approach to the semantics of evaluation would be expressed so that Minimal BASIC "+", to continue the example, is interpreted to mean implementation-floating-add, subject to the constraint that the value returned must be within ε of the standard-floating-add. Thus both add operations would be evaluated, as would ε, and the computation would be allowed to continue (i.e., would be correct) only if the constraint were met. The value carried through the calculation would be that of the implementation-floating-add. This might be given in SEMANOL(76) as follows:

```
    #DF floating-add(x,y)⇒implementation-floating-add(x,y)#IF
             absolute-difference(standard-floating-add(x,y),implementation-
             floating-add(x,y))<epsilon('fad',x,y);
            ⇒error #OTHERWISE#.
```

In terms of the partioning of the SEMANOL(76) specification, the definitions of floating-add and standard-floating-add would appear in the common part while implementation-floating-add (and its subsidiary definitions) would be part of the implementation dependent portion of the specification. As suggested, we hope that epsilon can be given by a generalized function and so be part of the specification; if such generalization cannot be achieved, then the value of epsilon would appear in the implementation dependent part of the specification.

Thus a SEMANOL(76) specification can make very precise what the operation denoted by "+" means, for instance. Conventional specifications leave such basic operations undefined, although the informal expectation is that addition will be done using the instruction and data representations natural to the host computer. Whether this notion should be extended to built-in functions (e.g., SIN, COS, SQRT) is uncertain since it appears that language designers often intend that certain functions be provided but really are uninterested in their specific operation (i.e., semantics). The question is thus seen as one of of deciding whether built-in functions are to be part of a programming language; if so, then language design must include an effort to define these functions much more carefully than has been true in the past. It seems SEMANOL(76) can readily provide the needed clarity of definition here too if it becomes appropriate to do so.

The second part of the problem deals with optimization. SEMANOL(76) specifies an exact sequence of steps to be performed in executing a computation. That is, the semantics are described in a totally deterministic manner. Thus the SEMANOL(76) method of describing computation fulfills the declaration of the ANSI/X3 committee that language standards should specify the order of expression evaluation. However, the committee did not rule out allowed deviations from that order, and the normal demands for computer efficiency, in any case, can be expected to result in continuing efforts to generate better code through the use of clever optimization techniques. To accomplish optimization demands that compiler writers must have some latitude in choosing how code is generated. In consequence, the ability to optimize means that a compiler is free to generate code which may produce answers which differ from those which would be obtained by strict adherence to the stated evaluation rules.

We would like to have a method of rigorously describing allowable optimization choices, and desire such a method of optimization specification to be reflected in the output from the SEMANOL(76) Interpreter. Our belief is that a partial solution to this problem can be developed by dealing with optimization in terms of alterations to the parse tree, and including within the standard specification a set of constraints that govern what an implementation may legally do in transforming the parse tree. Each implementation could be represented through inclusion of its optimization procedures, expressed in SEMANOL(76) or by an external function, within the total SEMANOL(76) specification. This optimization procedure would correspond to the optimization algorithms used by a given implementation. This optimizing procedure would be activated after the context-free parse (and context-sensitive testing) would be performed but before execution interpretation would be started. The revised (i.e., optimized) parse tree would then be tested to determine that it was still syntactically derivable from the original grammar (and met the context-sensitive tests), and that the transformations made were allowed under the optimization constraints that exist for the language being defined. This test for parse tree equivalence would be expressed in SEMANOL(76) code. Observe that optimization would be dealt with here in source language terms; this would be necessary if one is to characterize optimization for a variety of computers. This method does deal with a static form of operation reordering, such as a compiler might make; it does not consider any form of dynamic optimization that an interpreter or computer might be able to achieve during execution.

The fact that the allowable optimization transformations for a language could be given precisely would be very helpful. The availability of the SEMANOL(76) Interpreter, and so of a way to test implementation optimization algorithms, would also be useful. It would provide an excellent tool for developing optimization techniques that meet the standard requirements for optimization. It also would provide a method by which implementation conformance with its stated formal characteristics could be tested, a more precise testing mechanism than has heretofore been available with SEMANOL(76).

CONCLUSIONS AND RECOMMENDATIONS

This project was successful in preparing a SEMANOL(76) metalanguage specification of the Minimal BASIC programming language that was complete, precise (by the nature of the SEMANOL(76) metalanguage), and well tested. The extensive testing of the specification that was done, by means of the SEMANOL(76) Interpreter computer program, insured the soundness of the Minimal BASIC specification product. The specification process itself caused many questions about the X3J2 draft proposal to be identified and reported; the application of SEMANOL(76) thereby provided a useful critical analysis to the conventional Minimal BASIC definition document. This project thus met its objectives, and so created a useful formal specification of Minimal BASIC.

This project was also able to deal somewhat with the problem of programming language dependency upon implementation factors. This effort is evident in the form given to the SEMANOL(76) specification of Minimal BASIC, which was written so as to be especially revealing about those aspects of the programming language that are meant to be dependent upon the implementation of each language processor. In addition, a prospective means of formally characterizing machine arithmetic and optimization options was developed. While this development was promising, it was unexpected and so could not be fully applied to Minimal BASIC because of contract limitations.

There are two natural extensions of this work that should be considered for future activity. The first extension is to update the Minimal BASIC specification we have delivered to agree with the ANSI standard that is eventually adopted. As we have observed in this report, the specification prepared in this project reflects the X3J2 committee draft proposal of January 1976. At least one other version has been produced, in December 1976, that changed the Minimal BASIC programming language slightly; other changes and clarifications remain possible. These changes ought to be incorporated into the formal specification we have provided with SEMANOL(76). The second extension is to attempt to apply the proposed methods of formally dealing with arithmetic and optimization standards for language processors. Because Minimal BASIC is a somewhat simpler language than most, and because the matter of implementation dependencies has already been faced in the draft proposal, Minimal BASIC appears to be an excellent vehicle for attacking this difficult theoretical and practical problem. Taken together, these two extensions would provide a uniquely comprehensive specification for Minimal BASIC.

ADDENDUM

Operating Instructions for the SEMANOL(76) Specification of BASIC

Before any Minimal BASIC programs can be interpreted using the SEMANOL(76) Executer and the SEMANOL(76) specification of Minimal BASIC, the user must be able to reference, from his working directory, both the translated SEMANOL(76) specification of Minimal BASIC (SIL code) and the SEMANOL(76) Executer. In a one-time operation, this is done by creating links from the user's directory to these files. The link to the SIL code can be created with the following Multics command:

```
        lk    > user _ dir _ dir > 5550c0840 > Hart > file 22.basic.76
```

Since many different functions can be performed by the SEMANOL(76) Executer, many links from the user's directory to the Executer need to be created. To simplify this task for the user, an exec_com segment has been written which, if executed, will generate all of the necessary links. Therefore, the user needs first, to generate a link to this exec_com segment, and, second, to execute the segment. This can be done with the following sequence of commands:

```
        lk    > user_ dir _ dir > 5550c0840 > ERAnderson >
           new_interp > check > executer_link.ec

           ec executer_link
```

Once the appropriate links have been constructed, the Minimal BASIC SIL file must be initialized and loaded. This is done with the following command:

```
           semanol  file22.basic.76
```

This step needs to be performed only once during a single user process. After it is completed, as many Minimal BASIC programs as desired can be successively interpreted. The output from this command consists of several pairs of lines of the form

```
           scomp called

           scomp returns
```

The Minimal BASIC program to be interpreted should be contained in a Multics segment, for example one named file.prog. Each line in the program must be terminated by a line feed ([LF]) character; this character is inserted naturally by the normal editing process. If there is any input data, that is, information to be read by an INPUT statement, it should be contained in a second Multics segment, e.g., file.data. Again, each line should be terminated by a line feed. Then, to run the Executer, the following command should be used:

```
run file.prog file.data
```

If there is no input data, the appropriate command is

```
run file.prog
```

The run command can be executed as many times as desired, using different values of file.prog and/or file.data; each such execution causes interpretation of a single BASIC program.

The output from a run command consists of the results of executing any PRINT statements encountered in the BASIC program. In addition, if any INPUT statements are encountered, a question mark (?) is output, followed by a space and the line that is to be input. This mimics the interactive protocol. If any errors are encountered while processing the specification with the current BASIC program as input,messages are output to the user. Such errors can indicate either an erroneous BASIC program or a bug in the SEMANOL(76) specification of BASIC. The former kind of error is designated by a descriptive message (e.g., "subscript out of bounds"); if the error is fatal, the following four lines also are output:

```
execution terminated
mstop called
in fatal-error at location 23:level 7
STOP
```

An error in the SEMANOL(76) specification of BASIC, on the other hand, is indicated by the following line:

```
merr called
```

Finally, on termination, a few lines are output by the Executer.

The SEMANOL(76) Exeucter has several additional capabilities, which are used mainly for debugging language specifications written in SEMANOL(76). For the most part, these features will be discussed in the Executer documentation. However, one feature that may be of some interest now is the ability to trace the evaluation of a SEMANOL specification (of Minimal BASIC, in this case). The trace feature itself has many different options, not discussed here, but to turn the full trace on, the following command suffices:

```
tron  #CONTROL
```

This command should be given following the semanol command, but before a run command. It should be noted that the amount of trace generated is quite extensive, even for the simplest BASIC programs, so it is not recommended that this feature be used in the normal course of operation. To turn the full trace off, if it is on, the following command should be given preceding any run command:

```
troff  #CONTROL
```

METRIC SYSTEM

BASE UNITS:

Quantity	Unit	SI Symbol	Formula
length	metre	m	...
mass	kilogram	kg	...
time	second	s	...
electric current	ampere	A	...
thermodynamic temperature	kelvin	K	...
amount of substance	mole	mol	...
luminous intensity	candela	cd	...

SUPPLEMENTARY UNITS:

plane angle	radian	rad	...
solid angle	steradian	sr	...

DERIVED UNITS:

Acceleration	metre per second squared	...	m/s
activity (of a radioactive source)	disintegration per second	...	(disintegration)/s
angular acceleration	radian per second squared	...	rad/s
angular velocity	radian per second	...	rad/s
area	square metre	...	m
density	kilogram per cubic metre	...	kg/m
electric capacitance	farad	F	A·s/V
electrical conductance	siemens	S	A/V
electric field strength	volt per metre	...	V/m
electric inductance	henry	H	V·s/A
electric potential difference	volt	V	W/A
electric resistance	ohm		V/A
electromotive force	volt	V	W/A
energy	joule	J	N·m
entropy	joule per kelvin	...	J/K
force	newton	N	kg·m/s
frequency	hertz	Hz	(cycle)/s
illuminance	lux	lx	lm/m
luminance	candela per square metre	...	cd/m
luminous flux	lumen	lm	cd·sr
magnetic field strength	ampere per metre	...	A/m
magnetic flux	weber	Wb	V·s
magnetic flux density	tesla	T	Wb/m
magnetomotive force	ampere	A	...
power	watt	W	J/s
pressure	pascal	Pa	N/m
quantity of electricity	coulomb	C	A·s
quantity of heat	joule	J	N·m
radiant intensity	watt per steradian	...	W/sr
specific heat	joule per kilogram-kelvin	...	J/kg·K
stress	pascal	Pa	N/m
thermal conductivity	watt per metre-kelvin	...	W/m·K
velocity	metre per second	...	m/s
viscosity, dynamic	pascal-second	...	Pa·s
viscosity, kinematic	square metre per second	...	m/s
voltage	volt	V	W/A
volume	cubic metre	...	m
wavenumber	reciprocal metre	...	(wave)/m
work	joule	J	N·m

SI PREFIXES:

Multiplication Factors	Prefix	SI Symbol
1 000 000 000 000 = 10^{12}	tera	T
1 000 000 000 = 10^{9}	giga	G
1 000 000 = 10^{6}	mega	M
1 000 = 10^{3}	kilo	k
100 = 10^{2}	hecto*	h
10 = 10^{1}	deka*	da
0.1 = 10^{-1}	deci*	d
0.01 = 10^{-2}	centi*	c
0.001 = 10^{-3}	milli	m
0.000 001 = 10^{-6}	micro	μ
0.000 000 001 = 10^{-9}	nano	n
0.000 000 000 001 = 10^{-12}	pico	p
0.000 000 000 000 001 = 10^{-15}	femto	f
0.000 000 000 000 000 001 = 10^{-18}	atto	a

* To be avoided where possible.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

AMERICAN REVOLUTION BICENTENNIAL 1776-1976